Helena Clifford (261104576)
Léanne Ricard (261118883)

ECSE 429 Project Part B Report

Gherkin Tests

Todos

The gherkin tests were chosen to comprehensively validate the core functionalities of the To-Do API, ensuring that it correctly handles the creation, retrieval, updating, deletion and linking of to-do items. We structured the test suite to align with real-world use cases, focusing on both normal and edge cases to confirm that the API behaves as expected under different conditions. Each feature file corresponds to a major API functionality, allowing for modular and maintainable testing.

We designed the create_todo.feature to verify that users can successfully create to-do items with valid inputs while also handling scenarios where optional fields, such as descriptions, are omitted. Additionally, I included error-handling tests to ensure the API correctly rejects invalid requests, such as missing required fields. The delete_todo.feature ensures that users can remove to-do items and validate that deleted items are no longer accessible. Similarly, the get_todo.feature confirms that users can retrieve specific to-do items by ID, while also testing for appropriate error responses when requesting a non-existent item.

The update_todo.feature was included to verify that users can modify existing to-do items, ensuring that updates are applied correctly while maintaining data integrity. This feature also tests invalid updates, such as attempting to modify a non-existent item or providing improperly formatted data. Lastly, I created link_todo.feature to test the association between to-dos and projects, ensuring that to-do items can be assigned to projects and verifying that the API properly manages these relationships.

By selecting these tests, I aimed to cover both the expected behaviour of the API and its handling of invalid inputs, ensuring that it returns appropriate status codes and error messages. The Gherkin structure follows a clear Given-When-Then format, making the tests readable and maintainable while aligning with the principles of Behaviour-Driven Development. These tests not only validate the correctness of individual endpoints, but also ensure that the API maintains consistency and reliability across different user actions.

Project

The story test suite was structured to comprehensively test the functionalities of the project management API, ensuring that users can effectively create, update, and delete projects while managing associated todos and categories. The tests were designed to align with real-world use cases, covering both expected user interactions and potential edge cases. The suite consists of five feature files, each validating a distinct aspect of the system. The create_project.feature file ensures that users can successfully create projects, handling both normal scenarios and edge cases such as duplicate project names. The update_project.feature tests the ability to modify project details, verifying that updates are applied correctly and ensuring that appropriate errors are returned when updating non-existent projects. The remove_project.feature focuses on validating project deletion, confirming that users can remove projects and handling scenarios where a project contains associated data that may impact deletion. The manage_todo.feature is responsible for testing the addition and removal of todos within a project, ensuring that tasks can be tracked properly and removed when necessary. Finally, the manage_category.feature verifies that users can assign and remove categories associated with projects, maintaining accurate project organization.

Each feature file consists of multiple scenarios, structured with a clear scenario name, followed by Given-When-Then steps that map to the corresponding step definitions in the test implementation. The example tables provide varied test data, allowing for multiple cases to be validated efficiently within a single scenario outline. The GET method is primarily tested within the create and update project feature files, ensuring that projects can be retrieved after creation and correctly modified. This structured approach maintains consistency across tests, making them adaptable for future expansions while providing thorough validation of API behavior.

Source Code Repository

Our source code repository follows a structured and modular layout separated by features and steps, and within these folders, separated by Gherkin scripts and the steps file.

```
/429Project_B
|── /features
|   |── projects

|   |   ├── manage_todo.feature
|   |   ├── create_project.feature
|   |   ├── remove_project.feature
|   |   ├── manage_category.feature
```

```
|   |   ├── update_project.feature
|   |── todos
|   |   ├── create_todo.feature
|   |   ├── delete_todo.feature
|   |   ├── get_todo.feature
|   |   ├── link_todo.feature
|   |   ├── update_todo.feature
|   |── steps
|   |   ├── project_steps.py
|   |   ├── todos_steps.py
|   |── environment.py
```

Steps

The step definitions serve as the bridge between the Gherkin feature files and actual API interactions, ensuring that each scenario executes correctly. The steps are implemented using Behave and structured in a modular way to maintain clarity and reusability.We created two steps file to ensure the methods defined would be tailored to the projects and todos testing while also making sure no definition was repeated to avoid errors.

Todos

The step definition file for to-dos was implemented to map each Gherkin test step to actual API requests and assertions. The structure follows a modular approach, ensuring that each test scenario in the Gherkin feature files is executed correctly. The file contains Given steps to set up preconditions, such as ensuring a to-do item exists before retrieval or deletion tests. The When steps handle API interactions, including sending Post, Get, Put and Delete requests to the /todos endpoint, dynamically passing parameters like titles, descriptions, and IDs. The Then steps focus on validating API responses, checking for expected status codes, and ensuring the returned data matches what was sent.

A key aspect of this implementation was managing context.todo_id to track created to-dos dynamically, avoiding hardcoded values that could lead to 404 errors when retrieving or deleting items. Additionally, robust error handling was implemented to validate API responses, ensuring that the correct status codes and error messages are returned for invalid inputs. The step definitions were designed to be reusable and scalable, allowing for future expansions without significant modifications. Overall, this file plays a crucial role in bridging the Gherkin test scenarios with real API interactions, providing a structured and reliable way to validate the To-Do API’s functionality.

Project

The step definition file for projects was designed to integrate the Gherkin test steps with actual API interactions, ensuring each scenario is executed accurately. The structure follows a modular approach, breaking down the test cases into Given, When, and Then steps. This organization allows for clear separation between setting up test conditions, performing API requests, and validating responses.

The Given steps establish preconditions such as ensuring a project or todo exists before running further tests. A key improvement was the introduction of context.project_id, which dynamically tracks project creation and retrieval. This avoids reliance on hardcoded project IDs, reducing the risk of test failures due to mismatched data. The step_project_already_exists function first checks whether a project with the given name exists by making a GET request to /projects. If the project is found, its ID is stored in context.project_id; otherwise, a new project is created via POST, and its ID is stored for later use. Similarly, the step_ensure_todo_exists function ensures that a todo exists within the current project before proceeding with further tests.

The When steps execute API actions such as creating, updating, and deleting projects or todos. These steps dynamically reference context.project_id, ensuring that the correct project is targeted in requests. For example, the step_create_todo function sends a POST request to /projects/{context.project_id}/tasks, ensuring the todo is associated with the correct project. The step_delete_project function similarly deletes the currently stored project, preventing test failures due to missing project references.

The Then steps validate API responses, checking for expected status codes and verifying returned data. The step_validate_response function confirms that the project ID and name returned by the API match the expected values, while the step_validate_todo_response_with_due_date function ensures that the response correctly includes a due date when specified. To verify project deletion, the step_validate_project_deletion function asserts that the API returns a 200 OK or 204 No Content response, confirming successful removal.

By implementing context.project_id, the step definitions effectively track dynamically created data, preventing common issues such as mismatched IDs or missing references. This modular approach ensures that the test suite remains maintainable and scalable, allowing for future expansions without significant modifications.

Testing

Todos

During the testing process, we followed a structured approach to validate the To-Do API by interactively running the Behave tests, identifying failures, and refining both the Gherkin scenarios and step definitions accordingly. Initially, we encountered issues such as 404 errors when retrieving or deleting to-dos, which indicated that the test setup was not correctly ensuring that the items existed before being accessed. To address this, we ensured that every test scenario involving modifying or deleting to-dos was preceded by a Given step that properly created and stored the todo_id. We also incorporated debugging techniques such as manually verifying API responses using curl commands and printing API outputs in Behave to compare actual responses against expected values.

Another significant challenge was handling invalid requests and ensuring the API responded with the correct error messages and status codes. Some scenarios initially failed due to unexpected response structures, where the 'error' key was missing from the response body. To resolve this, we updated the assertions to log API responses when tests failed, allowing us to adjust the text expectations based on the actual API behaviour. Additionally, for scenarios that required linking to-dos to projects, I encountered inconsistent relationships, leading to errors in retrieval and deletion tests. To fix this, I verified that projects were created before attempting to link them to the to-dos, preventing relationship-based failures.

Through this process, I was able to improve the robustness of the test suit by refining preconditions, API request handling, and response validation. By implementing context tracking for dynamically generated to-do IDs and handling test cleanup efficiently, I ensured that the test cases could run independently and consistently. This iterative bug-fixing approach significantly improved test reliability, making the Behave suit a strong validation tool for ensuring the API's correctness and stability.

Project

The testing process for project management functionalities was conducted iteratively, refining both the Gherkin feature files and their corresponding step definitions to align with API behavior. Initially, several issues emerged, such as assertion failures, incorrect response parsing, and inconsistencies in API error handling. These challenges required structured debugging and incremental improvements to ensure the reliability of the test suite.

One of the key adjustments involved ensuring that test scenarios properly set up preconditions. Many early failures resulted from attempting to interact with projects or todos that did not exist, leading to unnecessary 404 errors. To address this, Given steps were reinforced to verify resource existence before performing actions on them. This prevented retrieval and deletion tests from failing due to missing test data.

Additionally, API responses did not always match expectations, particularly when handling error cases. Some assertions failed due to missing or incorrectly formatted error messages. The step definitions were adjusted to handle these cases more flexibly, ensuring that error-handling tests remained robust while still verifying that the API returned meaningful responses.

Another improvement was refining assertions to correctly extract data from JSON responses. Some failures stemmed from improperly formatted checks that concatenated multiple response fields, leading to misleading assertion errors. This was corrected by improving the way the tests parsed and validated API output.

Ambiguities in step definitions were also resolved to streamline error handling and avoid redundant implementations. Instead of defining separate steps for every possible error case, a more generalized approach was taken to allow a single step definition to verify different error messages dynamically.

Through these refinements, the Behave test suite became a more effective tool for validating project-related API functionalities. The final implementation ensures that projects can be created, updated, linked with todos and categories, and deleted while handling edge cases appropriately. The structured approach to debugging and test improvement strengthened the reliability and maintainability of the test suite.

Findings

Todos

Through the execution of the Gherkin test suite, several key findings emerged regarding the behaviour and reliability of the To-Do API. The tests successfully confirmed that basic operations such as creating, retrieving, updating, and deleting to-dos, function correctly under normal conditions. However, initial failures exposed issues with how the API handled non-existent to-dos, as many GET and DELETE requests resulted in 404 Not Found errors when the expected to-do item was missing. This highlighted the importance of properly ensuring

pre-existing test data before executing these actions. Additionally, tests for creating to-dos with optional fields confirmed that the API correctly allows empty descriptions, while tests for invalid field inputs revealed some inconsistencies in error handling, where expected 400 Bad Request responses were sometimes returned without clear error messages.

Another major finding was related to the linking of to-dos to projects, where relationship-based actions initially failed due to missing preconditions. The API required that projects be created before attempting to associate them with to-dos, which was not always enforced in the early test cases. Additionally, while most API operations followed standard RESTful behaviour, some responses lacked consistent error structures, making it difficult to assert expected failure cases reliably. By refining the step definitions, dynamically storing created IDs, and adding better error handling assertions, the overall reliability of the test suite improved. The final test results indicate that the API performs well under expected conditions, but further improvements could be made in error response consistency and relationship management between to-dos and projects.

Project

The test results revealed that while the API correctly supports project creation, it struggles with deletion, updates, and relational operations. One key issue is the failure to enforce uniqueness constraints—attempting to create duplicate projects returned 201 Created instead of the expected 409 Conflict, allowing unintended duplicate entries.

Deletion failures were another major issue. The API often returned 200 OK when attempting to delete non-existent projects instead of 404 Not Found. Additionally, project deletions did not return proper confirmations, making it unclear whether the operation succeeded.

Managing todos and categories within projects also led to frequent failures, with 404 Not Found errors appearing even when projects were confirmed to exist. Error messages were inconsistent, sometimes returning vague responses like "Invalid GUID for 999 entity project" instead of "Project not found".

Project updates were not applied correctly, with 404 Not Found responses instead of the expected 200 OK, preventing modifications to project names and descriptions. Finally, JSON decoding errors occurred, particularly during deletions, indicating that the API sometimes returns empty responses instead of properly formatted JSON.